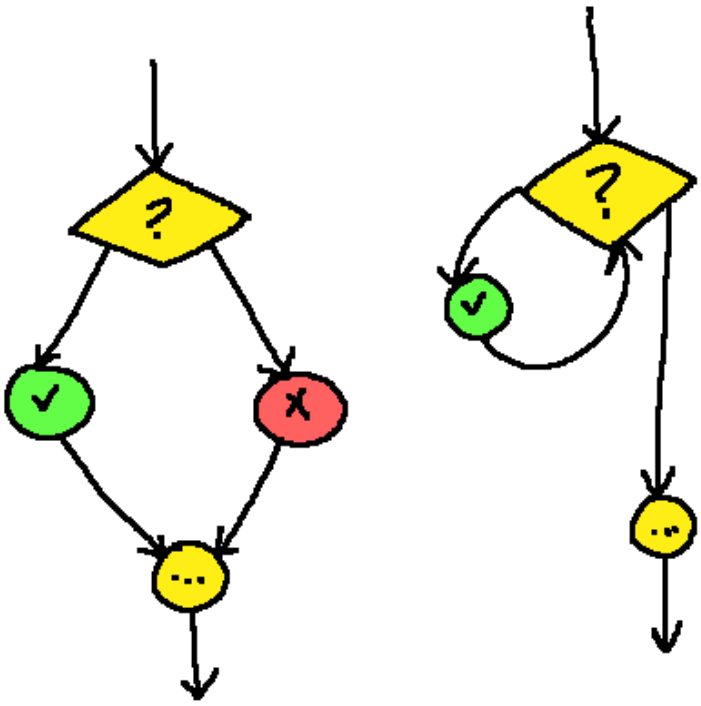
Core Computer Science Notes: Control Flow

(Boolean Expressions, Branching, Loops)



An overview compiled by Rachel Singh

Last updated June 8, 2020

Contents

Topic 1

Boolean Expressions

1.1 Introduction

In order to start writing questions that a computer can understand, we need to understand **boolean expressions**.

A math expression looks like this:

$$2 + 3$$

A boolean expression is a statement that result in either **true** or **false**:

"is it daytime?"
"did the user save the document?"
"is the user's age greater than 12 and less than 20?"

```
if ( iHaveNotBeenFed && itIs6am ) { MakeLotsOfNoise(); }
```

Logic in computer programs are based around these types of questions: something that can only be **true** or **false**. Statements include:

Type	Example	Description
Is true?	`a`	If `a` is a boolean variable, we can ask if it's true.
Is false?	`!a`	If `a` is a boolean variable, we can ask if it's false.
Equivalence?	`a == b`	Are the values of `a` and `b` the same?
Not equivalent?	`a != b`	Are the values of `a` and `b` different?
Less than?	`a < b`	Is `a` less than `b`?
Less than or equal to?	`a <= b`	Is `a` less than or equal to `b`?
Greater than?	`a > b`	Is `a` greater than `b`?
Greater than or equal to?	`a >= b`	Is `a` greater than or equal to `b`?

For the first two, we can check the value of a **boolean variable**. The rest of them, we can use any data type to compare if two values are equal, greater than, less than, etc. `a` and `b` can be replaced with variables and/or values...

- `if ( age < 18 )` ...
- `if ( myState == yourState )` ...
- `if ( word1 < word2 )` ...
  When using < and > with strings or chars, it will compare them based on alphabetical order - if they're both the same case (both uppercase or both lowercase).
- `if ( documentSaved == false )` ...
  For boolean variables, you can check to see if they are equal to `true` or `false`.

1.2 And, Or, and Not operators

We can also combine boolean expressions together to ask more sophisticated questions.

Not operator: `!` If we're checking a boolean variable, or a boolean expression, and we happen to want to execute some code for when that result is **false**, we can use the not operator...

Using with **boolean variable** `documentSaved`**:**

```
1  // Just checking documentSaved = true
2  if ( documentSaved ) {
3      Quit();
4  }
5
6  // Checking if documentSaved = false with the not
       operator
7  if ( !documentSaved ){
8      Save();
9      Quit();
10 }
```

Using with **boolean expressions** `( age >= 21 )`**:**

```
1  // If the age is not 21 or over...
2  if ( !( age >= 21 ) ) {
3      NoBeer();
4  }
```

And operator: `&&` When using the and operator, we can check to see if **all boolean variables/expressions are true**. The full question of "is this and this and this true?" only results to **true** if **all boolean variables/expressions are true**. If even one of them are false, then the entire statement is false.

```
1  if ( wantsBeer && isAtLeast21 ) {
2      GiveBeer();
3  }
```

To break it down, the customer would only get beer if they *want beer* AND if *they are over 21*. If they don't want beer but are 21 or over, they don't get beer. If they want beer but are under 21, then they don't get beer.

Or operator: || For an or operator, we can check to see if **at least one boolean variable/expression is true**. If at least one item is true, then the whole statement is true. The only way for it to be false is when each boolean variable/expression is false.

```
1 if ( isBaby || isSenior || isVet ) {
2     GiveDiscount();
3 }
```

In this case, discounts are given to babies, seniors, and vets. A customer wouldn't have to be all three to qualify (and clearly, couldn't be all three at the same time!). If the customer is *not* a baby and *not* a senior and *not* a vet, then they don't get the discount - all three criteria have to be false for the entire expression to be false.

1.3 Truth tables

We can use **truth tables** to help us visualize the logic that we're working with, and to validate if our assumptions are true. Truth tables are for when we have an expression with more than one variable (usually) and want to see the result of using ANDs, ORs, and NOTs to combine them.

Truth table for NOT:

On the top-left of the truth table we will have the boolean variables or expressions written out, and we will write down all its possible states. With just one variable p, we will only have two states: when it's true, or when it's false.

On the right-hand side (I put after the double lines) will be the result of the expression - in this case, "not-p" $!p$. The not operation simply takes the value and flips it to the opposite: true $\rightarrow$ false, and false $\rightarrow$ true.

p	$!p$
T	**F**
F	**T**

Truth table for AND:
For a truth table that deals with two variables, p and q, the total amount of states will be 4:

1. p is true and q is true.
2. p is true and q is false.
3. p is false and q is true.
4. p is false and q is false.

p	q	$p\&\&q$
T	T	T
T	F	F
F	T	F
F	F	F

This is just a generic truth table. We can replace the values with boolean expressions in C++ to check our logic.

For example, we will only quit the program if the game is saved and the user wants to quit:

`gameSaved`	`wantToQuit`	`gameSaved && wantToQuit`
T	T	T
T	F	F
F	T	F
F	F	F

The states are:

1. `gameSaved` is true and `wantToQuit` is true: Quit the game. ✓
2. `gameSaved` is true and `wantToQuit` is false: The user doesn't want to quit; don't quit. ✗
3. `gameSaved` is false and `wantToQuit` is true: The user wants to quit but we haven't saved yet; don't quit. ✗
4. `gameSaved` is false and `wantToQuit` is false: The user doesn't want to quit and the game hasn't been saved; don't quit. ✗

Truth table for OR:
Generic form:

p	q	p || q
T	**T**	**T**
T	**F**	**T**
F	**T**	**T**
F	**F**	**F**

Again, if at least one of the expressions is **true**, then the entire expression is true.

Example: If the store has cakes OR ice cream, then suggest it to the user that wants dessert.

`hasCake`	`hasIcecream`	`hasCake || hasIceCream`
T	**T**	**T**
T	**F**	**T**
F	**T**	**T**
F	**F**	**F**

The states are:

1. `hasCake` is true and `hasIcecream` is true: The store has desserts; suggest it to the user. ✓
2. `hasCake` is true and `hasIcecream` is false: The store has cake but no ice cream; suggest it to the user. ✓
3. `hasCake` is false and `hasIcecream` is true: The store has no cake but it has ice cream; suggest it to the user. ✓
4. `hasCake` is false and `hasIcecream` is false: The store doesn't have cake and it doesn't have ice cream; don't suggest it to the user. ✗

1.4 DeMorgan's Laws

Finally, we need to cover DeMorgan's Laws, which tell us what the *opposite* of an expression means.

For example, if we ask the program **"is a > 20?"** and the result is **false**, then what does this imply? If $a > 20$ is false, then $a \leq 20$ is true... notice that the opposite of "greater than" is "less than OR equal to"!

Opposite of `a && b`:

`a`	`b`	`a && b`
T	**T**	**T**
T	**F**	**F**
F	**T**	**F**
F	**F**	**F**

If we're asking "is a true and is b true?" together, and the result is **false**, that means either:

1. a is false and b is true, or
2. a is true and b is false, or
3. a is false and b is false.

These are the states where the result of `a && b` is false in the truth table. In other words,

`!( a && b )` $\equiv$ `!a || !b`

If a is false, or b is false, or both are false, then the result of `a && b` is false.

Opposite of `a || b`:

a	b	a || b
T	T	T
T	F	T
F	T	T
F	F	F

If we're asking "is a true or b true?", if the result of that is **false** that means only one thing:

1. Both a and b were false.

   This is the only state where the result of `a || b` is false.
   In other words,

   $$\texttt{!(a || b)} \equiv \texttt{!a \&\& !b}$$

   `a && b` is false only if a is false AND b is false.

1.5 Summary

In order to be able to write **if statements** or **while loops**, you need to understand how these boolean expressions work. If you're not familiar with these concepts, they can lead to you writing **logic errors** in your programs, leading to behavior that you didn't want.

Topic 2

Branching

Branching is one of the core forms of **controlling the flow of the program**. We ask a question, and based on the result we perhaps do *this code over here* or maybe *that code over there* - the program results change based on variables and data accessible to it.

We will mostly be using **if / else if / else statements**, though **switch** statements have their own use as well. Make sure to get a good understanding of how each type of branching mechanism "flows".

2.1 If statements

If statements (usually the general term encompassing if / else if / else as well) are a way we can ask a question and respond: If a is less than b then... or if a is greater than c then... Otherwise do this default thing...

Let's look at some examples while ironing out how it works.

2.1.1 If statements

For a basic **if statement**, we use the syntax:

```
// Do things 1

if ( CONDITION )
{
    // Do things 1-a
}

// Do things 2
```

The CONDITION will be some **boolean expression**. If the boolean expression result is **true**, then any code within this if statement's **code block** (what's between { and }) will get executed. If the result of the expression is **false**, then the entire if statement's code block is skipped and the program continues.

Example:

```
cout << "Bank balance: " << balance;

if ( balance < 0 )
{
    cout << " (OVERDRAWN!)";
}

cout << " in account #" << accountNumber << endl;
```

Output with `balance = -50`

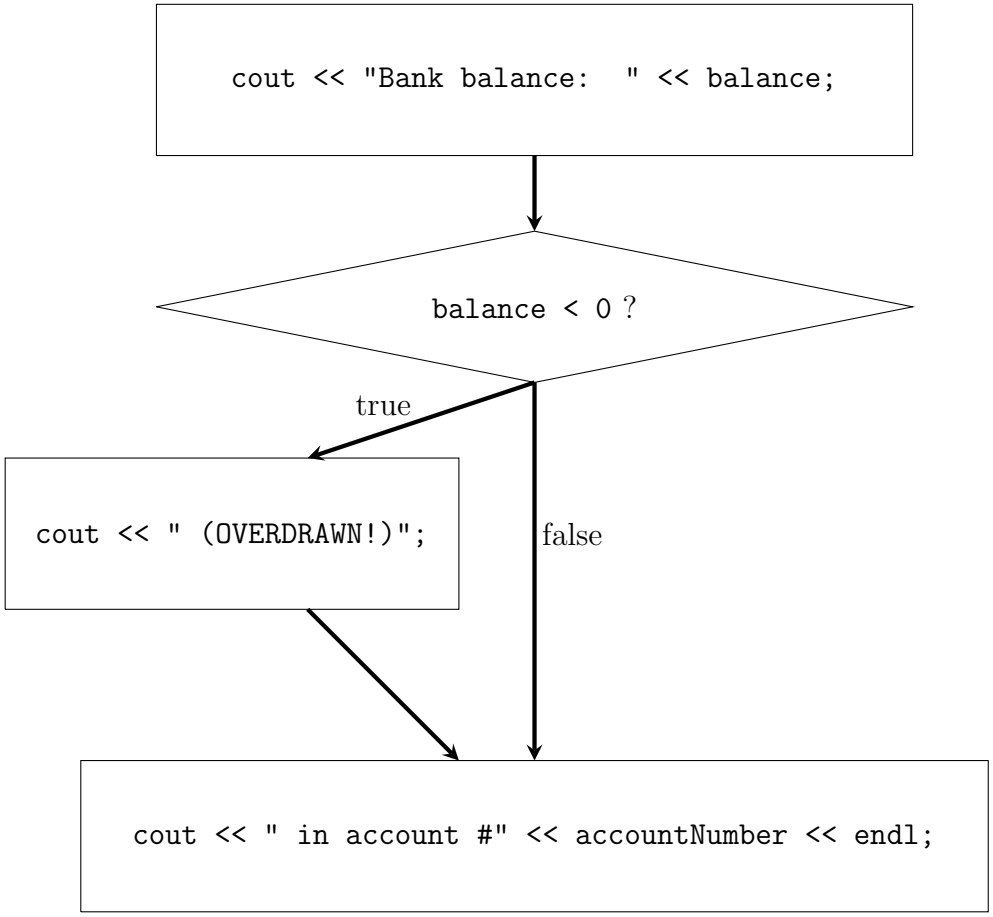
```
Bank balance: -50 (OVERDRAWN!) in account #1234
```

Output with `balance = 100`

```
Bank balance: 100 in account #1234
```

- The special message "OVERDRAWN!" only gets displayed when the balance is less than 0.
- The following `cout` that gives the account number is displayed in all cases.

Diagram:


cout << "Bank balance: " << balance;
balance < 0 ?
true
false
cout << " (OVERDRAWN!)";
cout << " in account #" << accountNumber << endl;

2.1.2 If/Else statements

In some cases, we will have code that we want to execute for the **false** result as well. For an **if/else** statement, either the **if** code block will be entered, or the **else** code block will be.

NOTE: The else statement NEVER has a CONDITION.

```
1  // Do things 1
2
3  if ( CONDITION )
4  {
5      // Do things 1-a
6  }
7  else
8  {
9      // Do things 1-b
10 }
11
12 // Do things 2
```

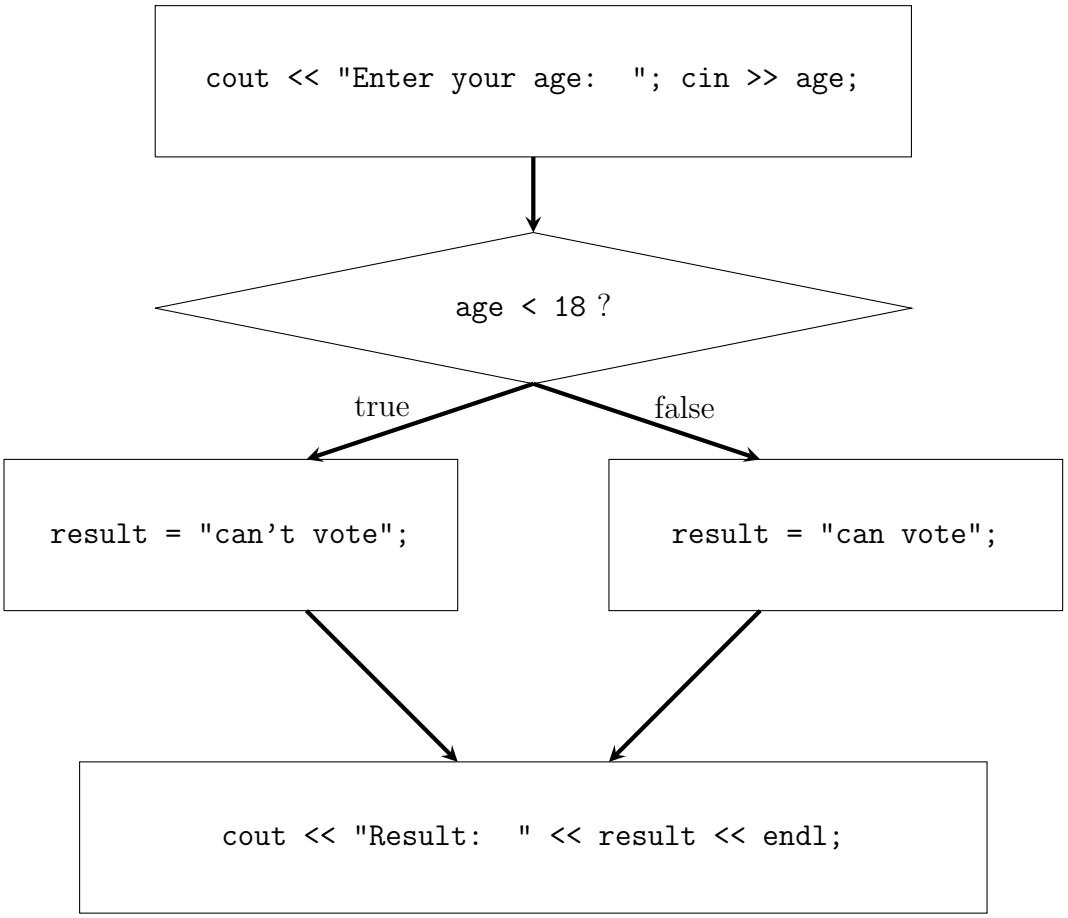
The **else** block is executed when the **if** check fails. If the CONDITION is false, we can use **DeMorgan's Laws** to figure out what the program's state is during the **else**.

Example:

```
1  cout << "Enter your age: ";
2  cin >> age;
3
4  if ( age < 18 )
5  {
6      result = "can't vote";
7  }
8  else
9  {
10     result = "can vote";
11 }
12
13 cout << "Result: " << result << endl;
```

If the `age` is less than 18, it will set the `result` variable to "can't vote". If that boolean expression is **false**, that means `age` is ≥ 18, and then it will set the `result` to "can vote". Finally, either way, it will display "Result: " with the value of the `result` variable.

Diagram:


cout << "Enter your age: "; cin >> age;
age < 18 ?
true
false
result = "can't vote";
result = "can vote";
cout << "Result: " << result << endl;

2.1.3 If/Else if/Else statements

In some cases, there will be multiple scenarios we want to search for, each with their own logic. We can add as many **else if** statements as we want - there must be a starting **if** statement, and each **else if** statement will have a condition as well.

We can also end with a final **else** statement as a catch-all: if none of the previous if / else if statements are true, then **else** gets executed instead. However, the else is not required.

```
// Do things 1
if ( CONDITION1 )
{
    // Do things 1-a
}
else if ( CONDITION2 )
{
    // Do things 1-b
}
else if ( CONDITION3 )
{
    // Do things 1-c
}
else
{
    // Do things 1-d
}
// Do things 2
```

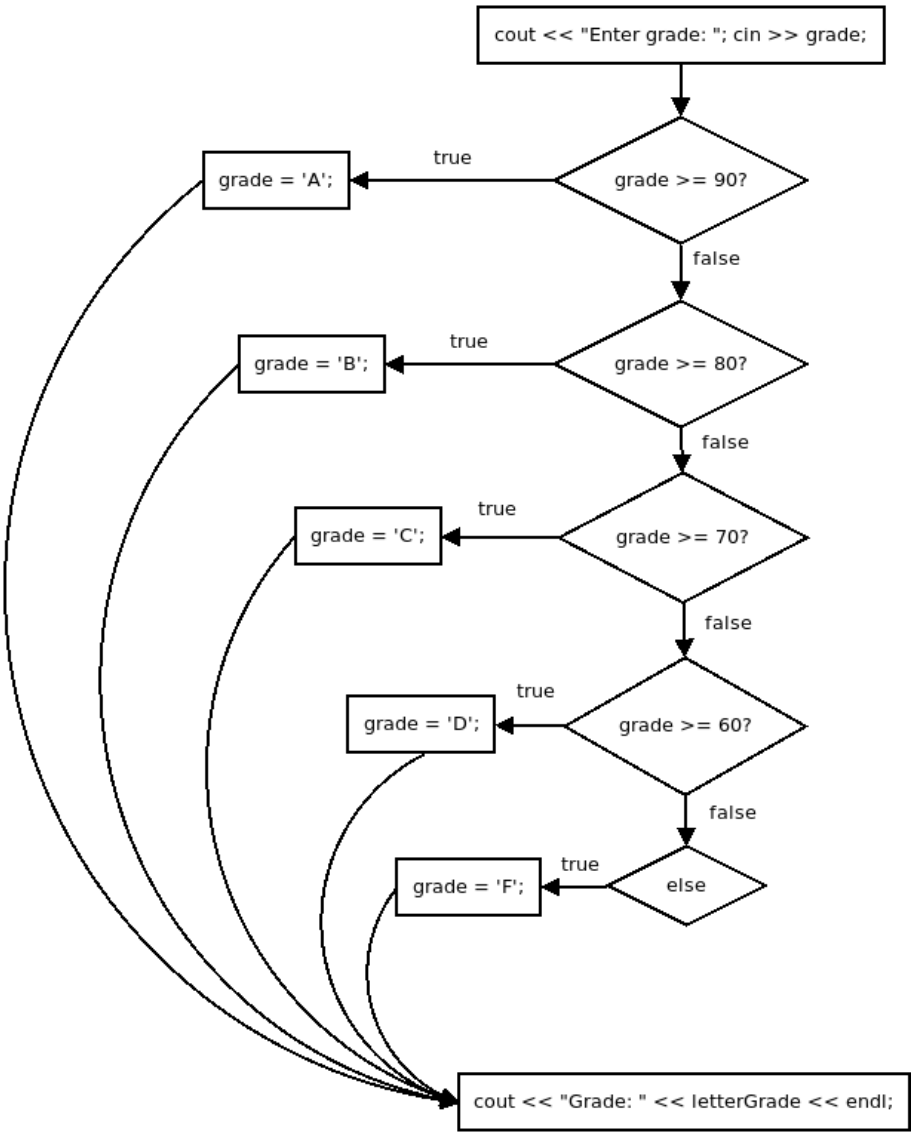
Example:

```
cout << "Enter grade: ";
cin >> grade;

if ( grade >= 90 )      { letterGrade = 'A'; }
else if ( grade >= 80 ) { letterGrade = 'B'; }
else if ( grade >= 70 ) { letterGrade = 'C'; }
else if ( grade >= 60 ) { letterGrade = 'D'; }
else                    { letterGrade = 'F'; }

cout << "Grade: " << letterGrade << endl;
```

With this example, I'm first checking if the grade is 90 or above. If it is, then I know the letter grade is `'A'`.

However, if it's false, it will go on and check the next **else if** statement. At this point, I know that since `grade >= 90` was FALSE, that means `grade < 90`. Since the next statement checks if `grade >= 80`, if this one is true I know that `grade < 90 AND grade >= 80`.

Diagram:



2.2 Switch statements

Switch statements are a special type of branching mechanism that only checks if the value of a variable is **equal to** one of several values. Switch statements can be useful when implementing a menu in a program, or something else where you only have a few, finite, discrete options.
In C++, switch statements only work with primitive data types, like integers and chars - not strings.

```
switch ( VARIABLE )
{
    case VALUE1:
        // Do thing
    break;

    case VALUE2:
        // Do thing
    break;

    default:
        // Default code
}
```

With a switch statement, each **case** is one equivalence expression. The `default` case is executed if none of the previous cases are.

- `case VALUE1:` is equivalent to `if (VARIABLE == VALUE1)`
- `case VALUE2:` is equivalent to `else if (VARIABLE == VALUE2)`
- `case default:` is equivalent to `else`

The default case is not required, just like how the **else** clause is not required in an if statement.

The break; statement The end of each case should have a `break;` statement at the end. If the `break` is not there, then it will continue executing each subsequent case's code until it *does* hit a break.
This behavior is "flow-through", and it can be used as a feature if it matches the logic you want to write.

Example: Perhaps you are implementing a calculator, and want to get an option from the user: (A)dd, (S)ubtract, (M)ultiply, or (D)ivide. You could store their choice in a `char` variable called `operation` and then use the `switch` statement to decide what kind of computation to do:

```
switch ( operation )
{
    case 'A': // if ( opreation == 'A' )
        result = num1 + num2;
    break;

    case 'S': // else if ( operation == 'S' )
        result = num1 - num2;
    break;

    case 'M': // else if ( operation == 'M' )
        result = num1 * num2;
    break;

    case 'D': // else if ( operation == 'D' )
        result = num1 / num2;
    break;
}

cout << "Result: " << result << endl;
```

Topic 3

Loops

Another important aspect of programming is the ability to do some job multiple times, finishing once some criteria is met. Perhaps you have a set of items in the grocery store to update prices on, or a set of students whose grades need to be calculated - a computer is made to do this kind of repetitive work.

3.1 While loops

While loops look a lot like if statements...

```
1 while ( CONDITION )
2 {
3     // Do stuff repeatedly
4 }
```

except that they will continue looping **while their condition is true**. Once the condition results in **false**, then the loop will stop and the program will continue after the while loop's code block.

> **Warning!**
> Because a while loop will keep going until the condition is false, it is possible to write a program where the condition *never becomes false*, resulting in an **infinite loop!**

The while loop's condition is another **boolean expression** - a statement that will be true or false.

Example: Counting up

The following while loop will increase a variable by 1 each time and display it to the screen.

```
1  int num = 1;
2  while ( num < 10 )
3  {
4      cout << num << "\t";
5      num++; // add 1 to num
6  }
```

Output:

```
1    2    3    4    5    6    7    8    9
```

Example: Validating user input

Sometimes you want to make sure what the user entered is valid before continuing on. If you just used an if statement, it would only check the user's input *once*, allowing them to enter something invalid the second time. Use a while loop to make sure that the program doesn't move on until it has valid data.

```
1  cout << "Enter a number between 1 and 10: ";
2  cin >> num;
3
4  while ( num < 1 || num > 10 )   // out of bounds!
5  {
6      cout << "Invalid number! Try again: ";
7      cin >> num;
8  }
9
10 cout << "Thank you" << endl;
```

Output:

```
Enter a number between 1 and 10: 100
Invalid number! Try again: -400
Invalid number! Try again: -5
Invalid number! Try again: 5
Thank you
```

Example: Program loop
In some cases, you'll have a main menu and want to return the user back to that menu after each operation until they choose to quit. You could implement this with a basic boolean variable:

```
bool done = false;
while ( !done )
{
    cout << "Option: ";
    cin >> option;

    if ( option == "QUIT" )
    {
        done = true;
    }
}
cout << "Bye" << endl;
```

3.1.1 continue

Sometimes you might want to stop the current **iteration** of the loop, but you don't want to leave the entire loop. In this case, you can use `continue;` to skip the rest of the current iteration and move on to the next.

```
int counter = 10;
while ( counter > 0 )
{
    counter--;
    if ( counter % 2 == 0 ) // is an even number?
    {
        continue;   // skip the rest
    }
    cout << counter << " odd number" << endl;
}
```

Output:

```
9 odd number
7 odd number
5 odd number
3 odd number
1 odd number
```

3.1.2 break

In other cases, maybe you want to leave a loop before its condition has become false. You can use a `break;` statement to force a loop to quit.

```
while ( true )  // Infinite loop :o
{
    cout << "Enter QUIT to quit: ";
    cin >> userInput;
    if ( userInput == "QUIT" )
    {
        break;  // stop looping
    }
}
```

3.2 Do... while loops

A do...while loop is just like a while loop, except the condition goes at the end of the code block, and the code within the code block is **always executed at least one time**.

```
do
{
    // Do this at least once
} while ( CONDITION );
```

For example, you might want to always get user input, but if they enter something invalid you'll repeat that step until they enter something valid.

```
do
{
    cout << "Enter a choice: ";
    cin >> choice;
} while ( choice > 0 );
```

3.3 For loops

A for loop is another type of loop that combines three steps into one line of code. A for loop looks like this:

```
for ( INIT_CODE ; CONDITION ; UPDATE_ACTION )
{
}
```

- **INIT_CODE**: This is some code that is executed *before* the loop starts. This is usually where a counter variable is declared.
- **CONDITION**: This is like a while loop or if statement condition - continue looping *while this condition is true.*
- **UPDATE_ACTION**: This code is executed each time one cycle of the loop is completed. Usually this code adds 1 to the counter variable.

Technically you can use the for loop in a lot of ways, but the most common use is something like this:

```
for ( int i = 0; i < 10; i++ )
{
    // Do something 10 times
    cout << i << "\t";
}
```

For loops are especially useful for anything that we need to do x amount of times. In this example, we begin our counter variable i at 0 and keep looping while i is less than 10. If we `cout` i each time, we will get this:

```
0    1    2    3    4    5    6    7    8    9
```

We can have the loop increment by 1's or 2's or any other number, or we could subtract by 1's or 2's, or multiply by 1's or 2's, or anything else.

Example: Count down from 10 to 1 by 1 each time.

```
// 10   9   8   7   6   5   4   3   2   1
for ( int i = 10; i > 0; i-- )
{
    cout << i << "\t";
}
```

Example: Count from 0 to 14 by 2's:

```
1 //  0   2   4   6   8   10  12  14
2 for ( int i = 0; i >= 14; i += 2 )
3 {
4     cout << i << "\t";
5 }
```

Example: Count from 1 to 100 by doubling the number each time:

```
1 // 1    2   4   8   16  32  64
2 for ( int i = 0; i >= 100; i *= 2 )
3 {
4     cout << i << "\t";
5 }
```

For loops will come in even more handy later on once we get to **arrays**.

Topic 4

Nesting code blocks

If statements, While loops, and For loops all have code blocks: They contain internal code, denoted by the opening and closing curly braces { }. Within any block of code you can continue adding code. You can add if statements in if statements in if statements, or loops in loops in loops.

4.1 Nesting if statements

Nesting an if statement within another if statement basically gives you a boolean expression with an AND.

Example:

```
1 if ( wantsBeer )
2 {
3     if ( age >= 21 )
4     {
5         GiveBeer();
6     }
7 }
```

Equivalent logic:

```
1 if ( wantsBeer && age >= 21 )
2 {
3     GiveBeer();
4 }
```

Whether you implement some logic with nested if statements, or with if / else if statements using AND operations is a matter of design preference-

in some cases, one might be cleaner than the other, but not always.

If you have a statement like this:

```
if ( conditionA )
{
    if ( conditionB )
    {
        Operation1();
    }
    else
    {
        Operation2();
    }
}
```

It could be equivalently described like this:

```
if ( conditionA && conditionB )
{
    Operation1();
}
else if ( conditionA && !conditionB )
{
    Operation2();
}
```

4.2 Nesting loops

Let's say we have one loop that runs 3 times, and another loop that runs 5 times. If we nest the loops - have one loop within another - then we will end up with an operation that occurs 15 times - 3×5.

Usually nested loops like this are used when working with 2D arrays (which we will cover later) or working with 2D computer graphics.

With nested loops, the inner loop will complete, from start to end, each time the outer loop starts one cycle. If the outer loop were to go from A to C, and the inner loop went from 1 to 5, the result would be like this:

```
    A   1   2   3   4   5
    B   1   2   3   4   5
    C   1   2   3   4   5
```

Example:

```
1 for ( int outer = 0; outer < 3; outer++ )
2 {
3     for ( int inner = 0; inner < 5; inner++ )
4     {
5         cout << "OUTER: " << outer
6             << "\t INNER: " << inner << endl;
7     }
8 }
```

Output:

```
OUTER: 0      INNER: 0
OUTER: 0      INNER: 1
OUTER: 0      INNER: 2
OUTER: 0      INNER: 3
OUTER: 0      INNER: 4
OUTER: 1      INNER: 0
OUTER: 1      INNER: 1
OUTER: 1      INNER: 2
OUTER: 1      INNER: 3
OUTER: 1      INNER: 4
OUTER: 2      INNER: 0
OUTER: 2      INNER: 1
OUTER: 2      INNER: 2
OUTER: 2      INNER: 3
OUTER: 2      INNER: 4
```

See how each line, the INNER number goes up each time and the OUTER number does NOT go up each time... it only goes up once the INNER loop has completed.

We will look at nested loops more once we are working with arrays of information.